Redis Cache Stampede Prevention in Node.js

A Production Guide to Request Coalescing, TTL Jitter, and Safe Distributed Locks

---

Prepared by Endurance Softwares
Cache Reliability Engineering • Node.js & Redis
www.endurancesoftwares.com

What Is a Cache Stampede?

A cache stampede, also known as a thundering herd, occurs when many requests discover the same missing or expired cache entry at once and independently recompute it. A popular product page, a tenant configuration, a pricing catalogue, or any expensive aggregate can turn a single expiry event into hundreds of simultaneous database queries or upstream API calls. The cache itself stays healthy while the origin system becomes the bottleneck.

This risk is not limited to high-traffic spikes. Synchronized TTLs can expire a large group of keys together after a deployment, a bulk import, or a mass invalidation. A Redis restart, an aggressive eviction policy, a cold regional rollout, or a failed refresh worker can create the same failure shape. For teams evaluating dependency resilience more broadly, our guide to Node.js circuit breakers for resilient dependencies covers how to protect the origin layer itself once cache coordination is in place.

The correct objective is broader than hit rate alone: bound origin concurrency for each logical resource, preserve an acceptable stale response when possible, and recover cleanly without permanent locks or poisoned data.

Choosing a Stampede-Control Strategy

Different techniques suit different failure semantics, and in practice they compose rather than compete.

Technique	Best Fit	Trade-off
In-process single-flight	Duplicate requests inside one Node.js instance	Cannot coordinate across replicas or regions
Randomized TTL (jitter)	Preventing many keys from expiring together	Does not protect one extremely hot key
Stale-while-revalidate	Read-heavy data with a safe stale window	Clients may briefly receive older data
Per-key Redis lock	Coordinating a refresh across application replicas	Needs lease expiry and ownership-safe release
Proactive refresh	Predictably hot keys with measurable refresh cost	Can refresh data nobody will request

A practical default combines in-process single-flight, TTL jitter, and stale-while-revalidate. A distributed per-key lock is worth the added operational complexity only when duplicate refresh work across replicas is genuinely costly. Redis should be treated as a coordination layer, not the final safety boundary — the origin still needs its own concurrency limit, timeout, and circuit breaker.

Cache-Aside Data Flow

1. Read a versioned cache key.

2. Return a fresh value immediately when one exists.
3. If the value is stale but still usable, return it and let one caller refresh in the background.
4. If no usable value exists, let one caller load the origin while peers wait briefly or fail predictably.
5. Write a payload with explicit freshness timestamps, using a randomized Redis expiry longer than the stale window.

Freshness should live inside the cached value itself rather than being fully delegated to the Redis TTL. Redis expiry then becomes cleanup, while the application distinguishes between fresh, stale-but-servable, and absent data.

Implementing Request Coalescing in Node.js

The pattern below uses the official node-redis client and separates the core caching logic from framework routing. Always validate the origin result before caching it, use a schema-versioned key, and never cache an authorization result under a key that omits the user or tenant boundary.

```
const inFlight = new Map();

function singleFlight(key, work) {
const running = inFlight.get(key);
if (running) return running;
const promise = work().finally(() => inFlight.delete(key));
inFlight.set(key, promise);
return promise;
}

function jitter(seconds) {
return Math.max(1, Math.round(seconds * (0.9 + Math.random() * 0.2)));
}

async function loadCached(key, origin, freshSeconds = 60, staleSeconds = 300) {
const raw = await redis.get(key);
const cached = raw ? JSON.parse(raw) : null;
const now = Date.now();

if (cached && now < cached.freshUntil) return cached.value;

const refresh = () => singleFlight(key, async () => {
const value = await origin();
const writtenAt = Date.now();
const envelope = {
value,
freshUntil: writtenAt + freshSeconds * 1000,
staleUntil: writtenAt + staleSeconds * 1000,
};
await redis.set(key, JSON.stringify(envelope), { EX: jitter(staleSeconds + 60) });
return value;
});

if (cached && now < cached.staleUntil) {
void refresh().catch(err => reportRefreshFailure(key, err));
return cached.value;
}
return refresh();
}
```

The in-process map only deduplicates within one process, but that remains valuable and inexpensive. Background refresh failures must surface to telemetry, and shutdown logic should stop accepting new work before closing the Redis connection — consistent with the pattern described in our graceful Node.js shutdown for zero-downtime deployments guide.

Coordinating Refreshes Across Replicas

When the origin operation is expensive, acquiring a per-key lease before refreshing prevents redundant work across application replicas. Redis documents the single-instance primitive as `SET key token NX PX duration`, where NX creates the lease only when absent and PX gives it an expiry. The token must be unique per acquisition, and release should only occur when the stored token still matches — preventing an old worker from deleting a newer worker's lease.

```
const token = crypto.randomUUID();
const leaseMs = 5000;
const acquired = await redis.set(lockKey, token, { NX: true, PX: leaseMs });

if (acquired === "OK") {
try {
return await refreshFromOrigin();
} finally {
await redis.eval(
'if redis.call("get", KEYS[1]) == ARGV[1] then ' +
'return redis.call("del", KEYS[1]) else return 0 end',
{ keys: [lockKey], arguments: [token] }
);
}
}
```

Lease duration should exceed the expected refresh time with margin, but every duration can still expire during a slow origin call, an event-loop pause, a failover, or a network partition. For cache regeneration this typically means occasional duplicate work, which is usually acceptable when writes are idempotent. This lightweight cache lease should never be reused for money movement, inventory allocation, or another correctness-critical workflow — those require database constraints, idempotency, and proper fencing.

Designing Keys, Invalidation, and Memory

Version keys instead of deleting an unknown universe. Keys such as `catalog:v3:tenant:42` allow a schema or serialization change to advance the version without mixing incompatible payloads. Every dimension that changes the answer — tenant, locale, permissions, query filters — should be part of the key, with large untrusted inputs hashed or normalized and cardinality capped.

Invalidate after the source of truth commits. For write-through invalidation, the database should be updated first, with the cache deleted or replaced only after commit. If a crash between those steps is unacceptable, invalidation should be published through a transactional outbox. The cache stays disposable; the database stays authoritative.

Configure Redis for cache memory behavior. A memory limit and eviction policy should be chosen from measured access patterns, with `INFO stats` used to monitor hits, misses, expired keys, and evictions. Teams building this out for a new service can involve Endurance Softwares for repository-supported custom software development and cloud infrastructure engineering, from cache

policy through deployment and operational validation.

Measuring Whether the Cache Protects the Origin

A high hit ratio can hide a severe stampede on one hot key. The full decision path deserves instrumentation with low-cardinality labels covering:

- Fresh hits, stale hits, hard misses, bypasses, and parse failures
- Single-flight joins, lease wins, lease contention, wait duration, and waiter timeouts
- Refresh latency, result size, failure rate, and origin calls per logical key
- Redis command latency, reconnects, timeouts, memory, evictions, and expired keys
- Database latency, pool saturation, error rate, and query concurrency during misses

Raw cache keys containing user IDs, search queries, or secrets should never be attached to metric labels; a safe namespace and sampled hash is sufficient for key-level diagnosis. Alerts should track origin amplification and stale age, not just Redis availability. For request-level correlation across cache, application, and origin calls, our request tracing and structured logging guide covers the underlying techniques in more depth.

Testing the Herd, Not Just the Happy Path

A unit test that performs a single miss cannot prove stampede control. Each state transition — fresh, stale, expired, evicted, malformed, and unavailable — deserves its own test, followed by concurrent integration tests against a counted origin stub:

1. Send many concurrent requests for one absent key and assert the allowed origin-call bound.
2. Repeat across multiple application processes to exercise the Redis lease.
3. Make the origin slower than the lease and verify duplicate refreshes do not corrupt data.
4. Crash the lease holder; confirm expiry enables recovery and waiters remain bounded.
5. Disconnect Redis mid-refresh and verify the chosen stale or failure behavior.
6. Expire thousands of keys together, then repeat with TTL jitter and compare origin concurrency.

Load tests should use a skewed key distribution, since uniform random requests miss the real risk that production traffic often concentrates on a small hot set.

Redis Cache Stampede Prevention Checklist

- check Define freshness and stale-on-error rules per data class
- check Version keys and include tenant and authorization dimensions
- check Coalesce duplicate work inside each Node.js process
- check Add TTL jitter to spread synchronized expiry
- check Serve stale data only inside an explicit safe window
- check Use unique lease tokens and ownership-safe release
- check Bound lease waits, retries, refreshes, and origin concurrency
- check Configure and monitor Redis memory and eviction
- check Validate payloads before caching and cap key cardinality
- check Test cold start, failure, contention, expiry, and recovery

Build Caching as a Reliability Feature

The right cache strategy follows the data's correctness rules, traffic shape, and failure budget. Endurance Softwares helps design and implement Node.js development, Redis-backed platforms, and production observability so caching stays a reliability feature rather than hidden operational debt.

Read the full guide: endurancesoftwares.com/blog/redis-cache-stampede-prevention-nodejs-guide-2026

Discuss your application architecture →

© 2026 Endurance Softwares. All rights reserved. www.endurancesoftwares.com